

Chapter 6

The Sequence Diagram

While the purpose of the state machine diagram presented in the last chapter is to model the *intra*-object behavior—that is, the life cycle of an object—in this chapter we look at the modeling of the *inter*-object behavior—that is, the interactions between the objects in a system.

An *interaction* specifies how messages and data are exchanged between interaction partners. The *interaction partners* are either human, such as lecturers or students, or non-human, such as a server, a printer, or executable software. An interaction can be a conversation between multiple persons—for example, an oral exam. Alternatively, an interaction can model communication protocols such as HTTP or represent the message exchange between humans and a software system—for example, between a lecturer and the student administration system when the lecturer publishes exam results. An interaction can also be a sequence of method calls in a program or signals such as a fire alarm and the resulting communication processes.

An interaction describes the interplay between multiple interaction partners and comprises a sequence of *messages*. The sending or receipt of a message can be triggered by the occurrence of certain events, for example, the receipt of another message, and can take place at specified times, for example, at 05:00. Predefined constraints specify any necessary preconditions that must be met for successful interactions. For example, continuing the communication process outlined above, the lecturer must be logged into the system before entering the students' grades.

In UML, you use *interaction diagrams* to specify interactions. In an interaction diagram, you always model a concrete scenario, meaning that the message exchange takes place within a specific context to fulfill a specific task. Interactions usually only describe a specific part of a situation. There are often other valid execution paths that the interaction

*Intra-object behavior
versus
inter-object behavior
Interaction
Interaction partner*

Message

Interaction diagram

diagram does not cover. Although data exchanged through the messages and processed or stored by the interaction partners can be represented in interaction diagrams, the purpose of modeling interactions is not to specify exactly how this data is to be manipulated. If required, you can add this type of information to interaction diagrams, but other diagrams such as the activity diagram (see Chapter 7) would take preference to model this information.

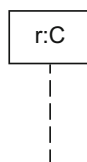
Use of interaction diagrams

Interactions offer a mechanism for describing communication sequences at different levels of detail, both for computer experts as well as for end users and decision-makers. Interaction diagrams are therefore used in various situations. For example, they are used to represent the interaction of a complete system with its environment. In this case, the system can be interpreted as a black box of which only the interfaces visible to the outside are known. You can also use interaction diagrams to model the interaction between system parts in order to show how a specific use case (see Chapter 3) can be implemented. In late design phases, you can use interaction diagrams to precisely model interprocess communication in which the partners involved must observe certain protocols. Interaction diagrams can also zoom in much further into the system to be realized and can model communication at class level, meaning that you can use them to model operation calls and inter-object behavior.

Of the four interaction diagrams offered by UML, the sequence diagram is the one most frequently used—often in an informal way to quickly present interaction sequences. However, in this chapter, we describe the elements of the sequence diagram in detail and examine how to apply them according to the UML standard. In Section 6.7 we briefly introduce the other three interaction diagrams and compare them to the sequence diagram.

6.1 Interaction Partners

Lifeline



In a sequence diagram, the interaction partners are depicted as *lifelines*. A lifeline is shown as a vertical, usually dashed line that represents the lifetime of the object associated with it (see Fig. 6.1). At the top end of the line is the head of the lifeline, a rectangle which contains an expression in the form `roleName:Class` (Fig. 6.1(c)). This expression indicates the name of the role and the class of the object associated with the lifeline. In the same way as for the object diagram (see Chapter 4.1 on page 50), one of the two names may be omitted. If you omit the class, you can omit the colon (Fig. 6.1(a)); however, if you specify only the class, the

colon must precede the class name (Fig. 6.1(b)). Thus you can define a sequence diagram at both instance level and class level.

You can also use other symbols for interaction partners instead of the rectangle, for example the stick figure that we saw in the use case diagram for actors (see Chapter 3).

In a sequence diagram, the use of the role concept allows more modeling flexibility than simple instances or classes. An object—that is, an instance of a class—can take on different *roles* over its lifetime. In our university system, it is quite conceivable that the person `helenLewis` is initially only a student, who then becomes a tutor, and finally a professor. With each new role, there are certain activities that Helen Lewis is no longer permitted to perform or no longer has to perform. However, there are other activities that she is now allowed to perform instead. If we considered only the class of the object `helenLewis` to reflect the different roles that the object can take, every time the role of the object changed we would have to delete the object and create a new one. Alternatively, the class would have to be changed dynamically.

Role

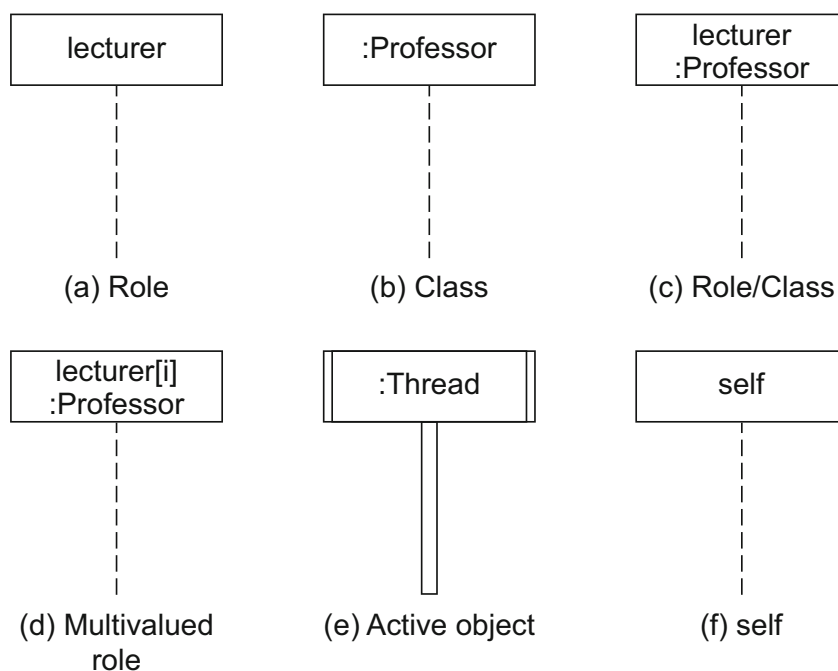


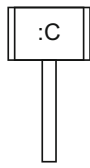
Figure 6.1
Types of lifelines

Roles can also be connected to more than one object. This type of role is referred to as a *multivalued role*. However, a lifeline may only represent one specific object. This object is selected by a *selector*. The selector is specified in square brackets between the role name and the colon. It can be formulated in any language, for example in natural language, pseudocode, or Java. In the example in Figure 6.1(d), the selector is simply a variable that acts as an index. To specify multiple objects of

Multivalued role
Selector

a role as independent interaction partners simultaneously, you must assign each object to a separate lifeline.

Active object



A lifeline can represent an *active object*. Active objects are used to model processes and threads. An active object has its own control flow, meaning that it can operate independently of other objects. The head of a lifeline that represents an active object has a double boundary on the left and right. A continuous bar is often used instead of the dashed line (see Fig. 6.1(e)).

In Figure 6.1(f), the head of the lifeline contains the name self. This is needed when a class spans a certain interaction context and is itself involved in the interaction.

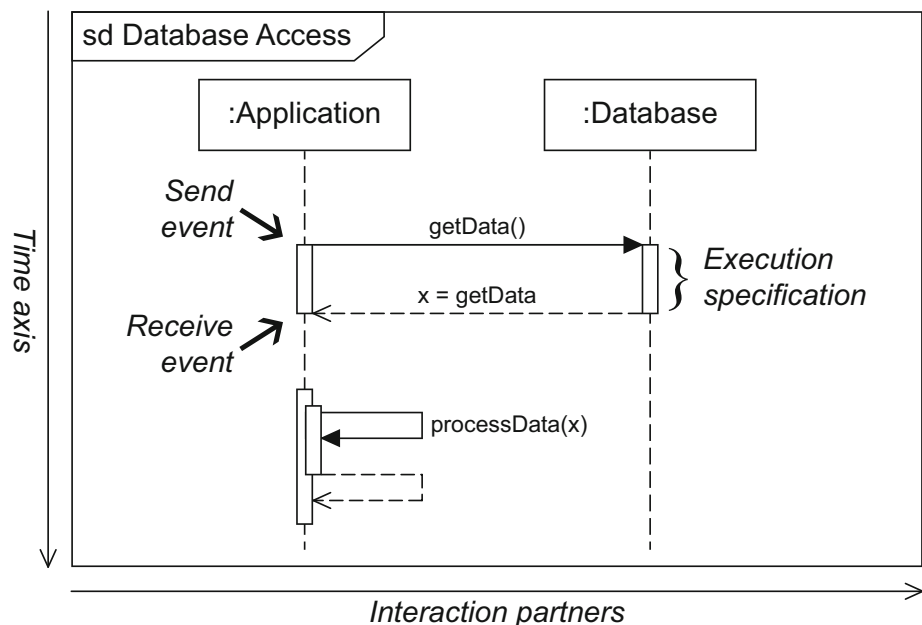
6.2 Exchanging Messages

The sequence diagram is a two-dimensional diagram (see Fig. 6.2). The interaction partners involved in the interaction are presented on the horizontal axis and should be arranged in a clear order. The vertical axis models the chronological order of the interaction. If the chronological order has not been explicitly set aside, an event further up on the vertical axis takes place before an event that is lower down on the vertical axis—provided both events refer to the same lifeline.

Event specification

In a sequence diagram, interactions are considered as a sequence of *event specifications*. Event specifications cover the sending and receipt of messages or the occurrence of time-based events such as a point in

Figure 6.2
Structure of a sequence diagram



time, for example. The vertical time axis determines the sequence of event occurrences on a lifeline, although this does not define the order of event occurrences on different lifelines. An order across multiple lifelines is only forced if messages are exchanged between the different lifelines. Unless specified otherwise, below we assume that the message transmission does not require any time, meaning that the send event at the sender and the receive event at the receiver take place at the same time. This allows us to present the traces more compactly because we do not have to consider sequences of send and receive events and can concentrate on sequences of messages. The chronological connection between a message a and a message b is expressed by the symbol \rightarrow . For example, $a \rightarrow b$ means that message a is sent before message b. [Figure 6.3](#) summarizes possible message sequences. If the send and receive events of two messages take place along the same lifeline, the chronological order of these events determines the order of the messages. In [Figure 6.3\(a\)](#), message a must always take place before message c, as the send event of a takes place before the send event of c. If two messages do not have any common interaction partners, the order of these messages is not specified. In [Figure 6.3\(b\)](#), this is the case for messages a and c. There are therefore two possible traces: $a \rightarrow c$ and $c \rightarrow a$. If a message b is inserted between a and c and this message forces a and c into a chronological order, the only possible trace is $a \rightarrow b \rightarrow c$ (see [Fig. 6.3\(c\)](#)).

Send event and receive event

Trace

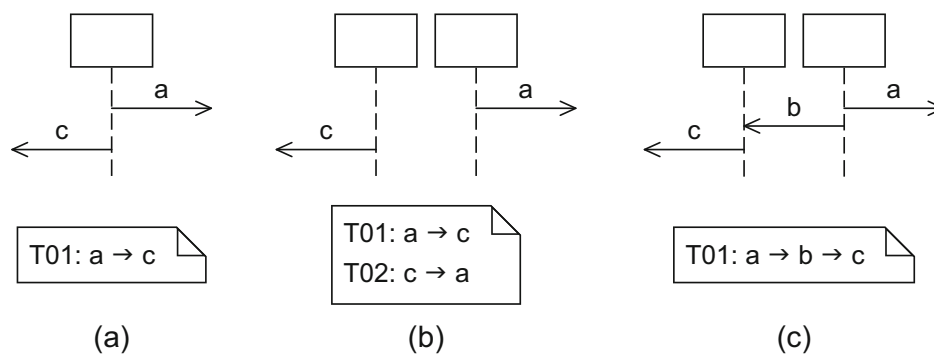


Figure 6.3
Sequences of messages and possible traces

The execution of behavior is indicated by two events that start and finish this execution on the same lifeline (see [Fig. 6.2](#)). This behavior is visualized with a bar and is referred to as an *execution specification*. The authors of the book UML@Work [23] differentiate between *direct* and *indirect* execution behavior. In the case of direct execution, the interaction partner affected executes the specified behavior itself; with indirect execution, the interaction partner delegates the execution to other interaction partners. Overlapping execution specifications are shown with overlapping bars. If an interaction partner sends a message to itself and

Execution specification

Direct versus indirect execution

Message to self

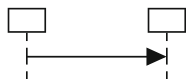
the associated execution specification is to be modeled explicitly, two bars of execution specifications are shown overlapping, whereby the message arrow points to the second bar, and in the case of a synchronous message, a dashed response arrow at the end of the execution leads back to the original bar (see `processData(x)` in Fig. 6.2).

You do not have to model execution specifications—they are optional and are mainly used to visualize when an interaction partner executes some behavior. Many UML tools draw the corresponding bars automatically as continuous bars from the first to the last message that affects an interaction partner. For reasons of clarity, in this book we generally do not show execution specifications in our sequence diagrams.

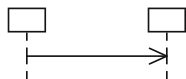
6.3 Messages

Message

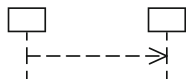
Synchronous message



Asynchronous message



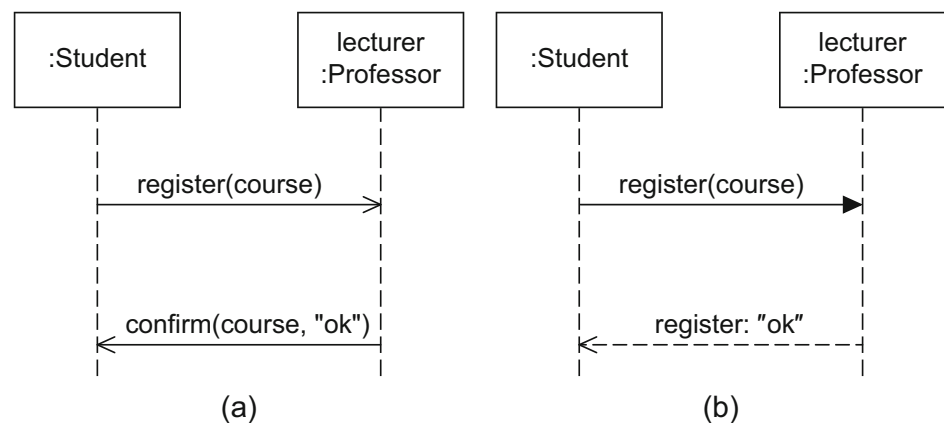
Response message



In a sequence diagram, a *message* is depicted as an arrow from the sender to the receiver. The type of the arrow expresses the type of communication involved. A *synchronous message* is represented by an arrow with a continuous line and a filled triangular arrowhead. An *asynchronous message* is depicted by an arrow with a continuous line and an open arrowhead. In the case of synchronous messages, the sender waits until it has received a *response message* before continuing. The response message is represented by a dashed line with an open arrowhead. If the content of the response message and the point at which the response message is sent and received are clear from the context, then the response message may be omitted in the diagram. In asynchronous communication, the sender continues after having sent the message. Two examples are shown in Figure 6.4.

Figure 6.4

Examples of (a) asynchronous and (b) synchronous communication



In both cases, a student is communicating with a professor in order to register for a course. In case (a), the registration is via e-mail, that is, asynchronous. The student does not explicitly wait for the receipt of the confirmation message. In case (b), the student registers with the professor personally and the communication is therefore synchronous. The student waits until receiving a response message.

Messages are identified by a name, with the optional specification of parameters and a return value (see Fig. 6.5). The parameters are separated by commas and are enclosed within parentheses. The return value can optionally be assigned to a variable as well. Thus, a message can be labeled with `var=m1:value`, whereby `var` is the variable to which the return value is to be assigned, `m1` specifies the name of the message, and `value` represents the actual return value.

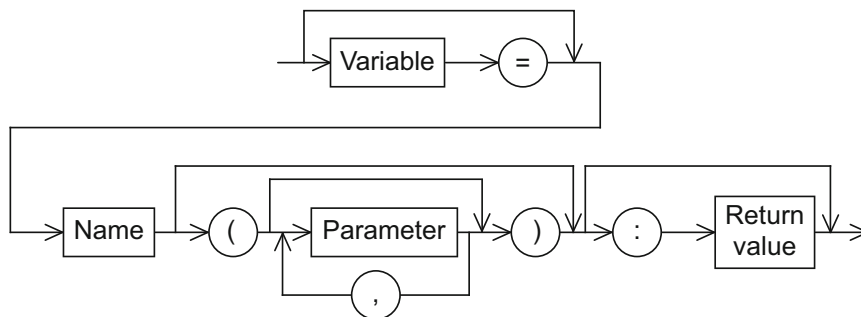


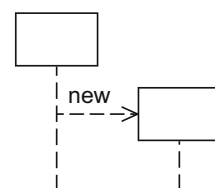
Figure 6.5
Syntax of the message specification

The receipt of a message by an object generally calls the corresponding operation specified in the class diagram (see Chapter 4). In principle, the passed arguments should be compatible with the parameters of the operation specification in the class diagram (see Fig. 4.8 on page 57). However, if you use parameter names to assign the values to the corresponding parameters, neither the number nor the order of the arguments has to match the parameters in the operation specification.

A *message for creating objects* is a special type of message. It is depicted by a dashed arrow with an open arrowhead that ends at the head of the lifeline associated with the object to be created. The arrow is labeled with the keyword `new` and corresponds to calling a constructor in an object-oriented programming language. For example, in Figure 6.6, a Professor creates a new ExamDate.

If an object is deleted during the course of an interaction, that is, a *destruction event* occurs, the end of the lifeline is marked with a large X (see Fig. 6.6). Otherwise a lifeline stretches to the lower end of the sequence diagram.

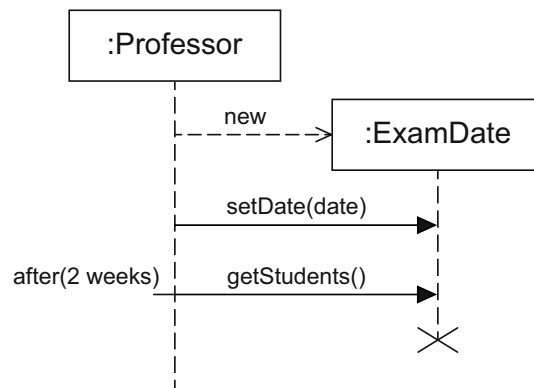
Create message



Destruction event



Figure 6.6
Creation of an object



Found message

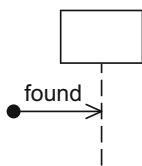
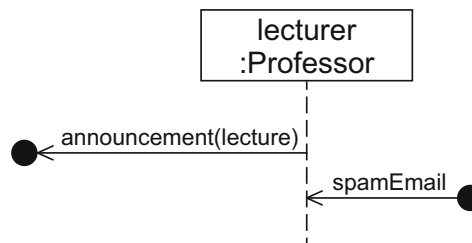
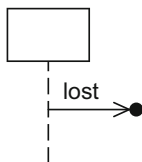


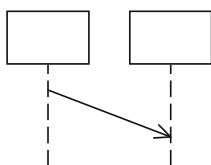
Figure 6.7
Example of lost and found messages



Lost message



Time-consuming message



Synonyms:

- *Time-consuming message*
- *Message with duration*

If the sender of a message is unknown or not relevant, you can express this with *found messages*. In this case, you use a black circle as source instead of specifying an interaction partner that sends the message. In [Figure 6.7](#), the sender of the message `spamEmail` is unknown.

The counterpart to the found message is the *lost message*. With this type of message, it is the receiver that is unknown or not relevant. The receiver is also noted as a black circle. The lecture announcement in [Figure 6.7](#) is sent to an arbitrary (and therefore unknown or irrelevant) receiver.

Up to this point, we have implicitly assumed that the messages are transmitted without any loss of time. Of course, this is not always the case. If you want to express that time elapses between the sending and the receipt of a message, you model the message as a diagonal line in the sequence diagram rather than a horizontal line. As the time dimension is represented vertically, this visualizes the duration required for the transmission of a message. This type of message is referred to as a *time-consuming message* or *message with duration*.

[Figure 6.8](#) shows an example scenario. A student enrolls for a study program in the student administration system. Within the next two to three days, the student receives a confirmation message affirming that the enrollment was successful. This confirmation is sent as a traditional letter and is therefore in transit for a few days before the student receives it.

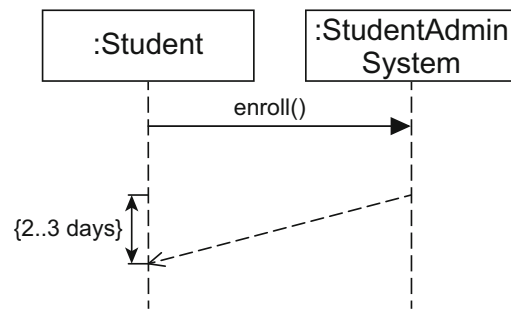


Figure 6.8
Example of a time-consuming message

6.4 Combined Fragments

In a sequence diagram, you can use *combined fragments* (operators) to model various control structures explicitly. This enables you to describe a number of possible execution paths compactly and precisely. Within a diagram, a combined fragment is represented by a rectangle, with the operator type specified by the respective keyword in a small pentagon in the upper left corner of this rectangle. UML offers 12 different types of operators. Depending on the type of the operator, it contains one or multiple operands which can in turn contain interactions, combined fragments, or references to other sequence diagrams. Different operands of an operator are separated from one another by horizontal, dashed lines. Gates describe the interfaces between a combined fragment and its environment (see Section 6.5.2).

In [23], the 12 different types of operators are split into three groups:

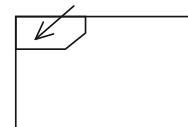
- Branches and loops
- Concurrency and order
- Filters and assertions

Table 6.1 provides an overview of the available operators with the corresponding keywords and their semantics. In the following, we refer to the different fragments according to their operator—for example, a combined fragment with an alt operator is referred to simply as an alt fragment.

Combined fragments can be nested arbitrarily, whereby a frame is specified for each fragment. Alternatively, nested fragments may share a frame. If this is the case, in the pentagon in the upper left corner of the frame, the corresponding keywords are specified separated by a space. The operator to the furthest left is assigned to the outermost fragment, and the operator to the furthest right is assigned to the innermost fragment (see Fig. 6.9).

Combined fragment

Operator



Operands

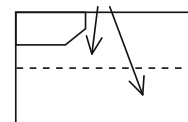
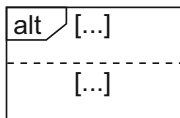


Table 6.1
Operators for combined
fragments

	Operator	Purpose
Branches and loops	alt	Alternative interaction
	opt	Optional interaction
	loop	Iterative interaction
	break	Exception interaction
Concurrency and order	seq	Weak order
	strict	Strict order
	par	Concurrent interaction
	critical	Atomic interaction
Filters and assertions	ignore	Irrelevant interaction parts
	consider	Relevant interaction parts
	assert	Asserted interaction
	neg	Invalid interaction

6.4.1 Branches and Loops

alt fragment



Alternative interaction

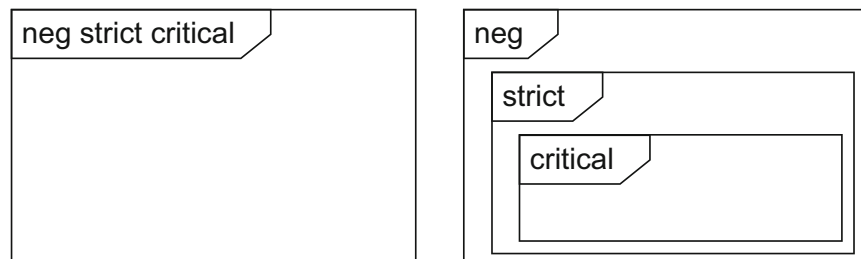
Guard

Predefined guard else

You can use an alt fragment to represent alternative sequences. An alt operator has at least two operands. Each operand represents an alternative path in the execution, which corresponds approximately to multiple cases in programming languages, for example, the `switch` statement in Java. Guards are used to select the path to be executed.

Each operand has a *guard*. A guard is a boolean expression enclosed within square brackets. If there is no guard, then `[true]` is assumed as the default value. If multiple guards are true simultaneously, this results in an indeterminism. In this case, there is no prediction regarding which operand is selected. This contrasts with the semantics of switch statements in common programming languages, in which the alternatives are usually processed from top to bottom. A special guard is `[else]`, which is evaluated as true if no other condition is fulfilled. If none of the guards evaluates to true, no operand is executed and the execution of the surrounding fragment continues. Figure 6.10 shows an example of the alt

Figure 6.9
Notation alternatives for
nested combined fragments



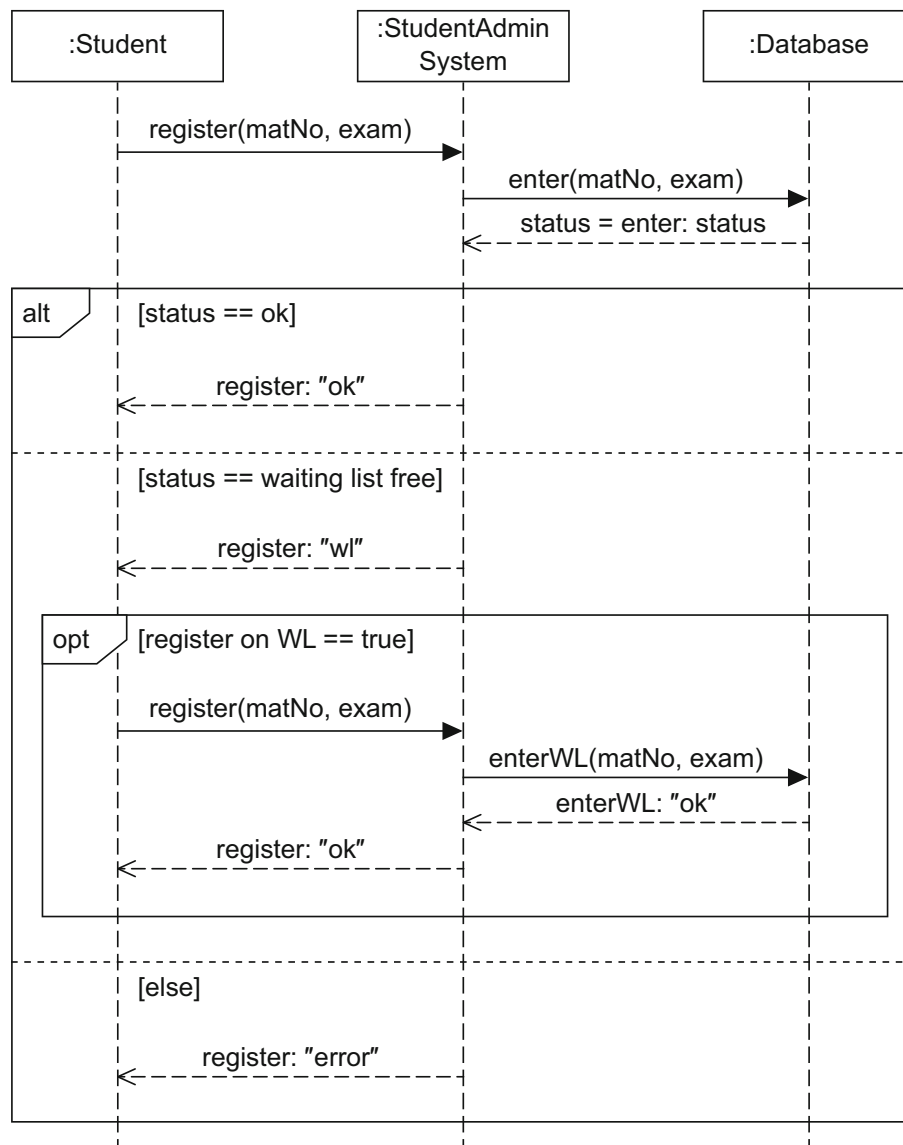
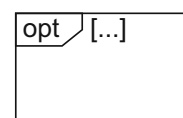


Figure 6.10
Example of an alt and an
opt fragment

fragment. When a student wants to register for an exam, the following cases can occur: (1) There are still places available and the student can register. (2) There is a place available on the waiting list. Then the student has to decide whether to go on the waiting list. (3) If there is no place available for the exam or on the waiting list for the exam, the student receives an error message and is not registered for the course.

The **opt** fragment corresponds to an **alt** fragment with two operands, one of which is empty. The **opt** operator thus represents an interaction sequence whose actual execution at runtime is dependent on the guard. In a programming language, this operator would be specified as an `if` statement without an `else` branch. Figure 6.10 illustrates the use of the **opt** fragment. If there is a place available on the waiting list, when registering for an assignment the student can decide whether to take the

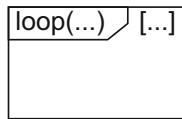
opt fragment



Optional interaction

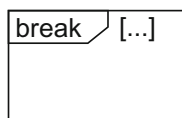
place on the waiting list. If the student wants to be on the waiting list, the student has to register for it.

loop fragment
Repeated interaction



You can use the loop fragment to express that a sequence is to be executed repeatedly. This combined fragment has exactly one operand. The keyword `loop` is followed by an optional specification of the number of iterations of the loop. This specification takes the form `(min..max)` or `(min,max)`, where `min` specifies the minimum number of iterations that the loop must go through and `max` denotes the maximum number of iterations. If `min` and `max` are identical, you can omit one of the two numbers and the dots. If there is no upper limit to the number of loop iterations, you only need to specify an asterisk `*`. In this case, the minimum number of iterations is assumed to be zero. If the keyword `loop` is not followed by any further specification of the number of iterations, `*` is assumed as the default value. If required, you can specify a guard, which is then checked for each iteration within the `(min,max)` limits. This means that the guard is evaluated as soon as the minimum number of iterations has taken place. If the underlying condition is not fulfilled, the execution of the loop is terminated even if the maximum number of executions has not yet been reached. [Figure 6.11](#) expands the example from [Figure 6.10](#) to include the system login that is necessary before a student can register for an assignment. The password must be entered at least once and at most three times, as reflected by the arguments of `loop`. After the first attempt, the system checks whether the password can be validated. If it can, that is, the condition `Password incorrect` is no longer true, execution of the interactions within the loop ceases. The system also exits the loop if the student enters the password incorrectly three times. This case is then handled further in the subsequent `break` fragment.

break fragment
Exception handling



The `break` fragment has the same structure as an `opt` operator, that is, it consists of a single operand plus a guard. If the guard is true, the interactions within this operand are executed, the remaining operations of the surrounding fragment are omitted, and the interaction continues in the next higher level fragment. The `break` operator thus offers a simple form of exception handling. For our example in [Figure 6.11](#), this means that if the password is entered incorrectly three times, the condition `incorrect password` is true. Thus the content of the `break` fragment is executed, meaning that an error message is sent to the student and the student is not allowed to register for the assignment. The remainder of the interaction after the end of the `break` fragment is skipped. After exiting the `break` operator, we are in the outermost fragment of the sequence diagram and therefore the execution of this sequence diagram is ended. If we were not in the outermost fragment, the sequence diagram would continue in the fragment at the next higher level.

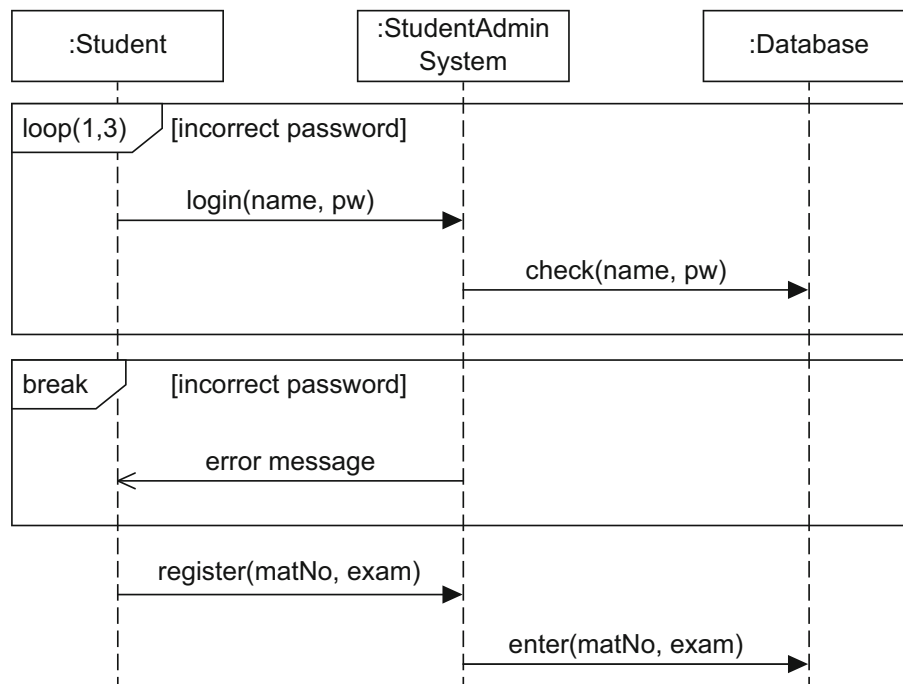


Figure 6.11
Example of a break and loop fragment

6.4.2 Concurrency and Order

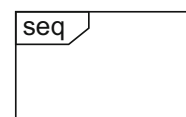
As already discussed, the arrangement of events on the vertical axis represents the chronological order of these events, provided there is a message exchange between the interaction partners involved. The combined fragments described below allow you to explicitly control the order of event occurrences.

The **seq** fragment represents the default order. It has at least one operand and expresses weak sequencing which is specified by the UML standard [35] as follows:

1. The ordering of events within each of the operands is maintained in the result.
2. Events on different lifelines from different operands may come in any order.
3. Events on the same lifeline from different operands are ordered such that an event of the first operand comes before that of the second operand.

We can use the **seq** fragment to group messages together with a **break** fragment. If the condition of the break fragment becomes true, the messages from the **seq** fragment that have not yet been executed are skipped and the execution of the sequence diagram continues in the surrounding fragment. [Figure 6.12](#) shows an example of this. A student wants to register for an exam. If there are no longer any places available for

seq fragment



Sequential interaction with weak order

Application of the seq fragments in conjunction with a break fragment

the desired date, the student makes a reservation for the next date (break fragment). In this case, the student is not examined by the lecturer and the execution of the sequence diagram continues outside the seq fragment. Irrespective of whether the registration was successful or not, the lecturer sends the message `info()` to the student.

Note that `seq` is the default order and usually does not have to be modeled explicitly. But in this case, without the explicit modeling of the `seq` fragment, the execution would have ended after the break fragment if `incorrect password` was true. This is due to the fact that after executing the content of a break fragment, the operations of the surrounding fragment are omitted. Without using `seq`, the surrounding fragment would have been the outermost structure of the diagram and thus the whole execution would have ended.

Figure 6.12
Example of a `seq`
fragment

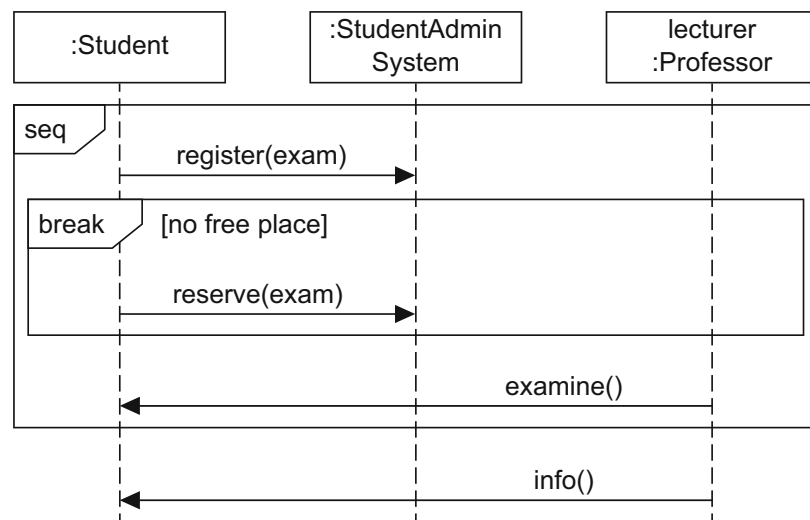
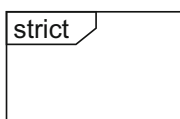


Figure 6.13 shows another sequence diagram together with all possible traces. As this diagram shows a weak order, the message `c` is not connected chronologically to messages `a` and `b` and can be interleaved with these messages. As `b` is sent by interaction partner `B` and `d` is also received by `B`, there is a chronological order between these two messages. In any case, `e` is the last message.

strict fragment



*Sequential interaction
with a strict order*

The strict fragment describes a sequential interaction with a strict order. The order of event occurrences on different lifelines between different operands is significant, meaning that even if there is no message exchange between the interaction partners, messages in an operand that is higher up on the vertical axis are always exchanged before the messages in an operand that is lower down on the vertical axis.

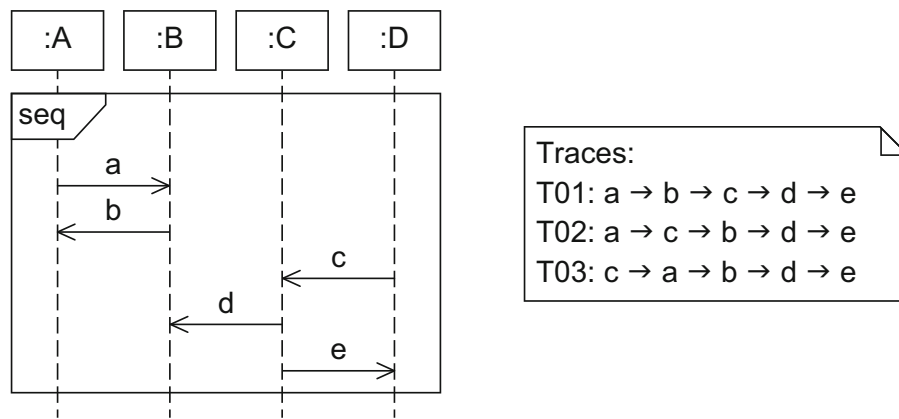


Figure 6.13
Traces in a seq fragment

In the example in [Figure 6.14](#), a lecturer only prints an exam when a student has registered for it. If the strict fragment were not specified, it would also be possible for the lecturer to print the exam before a student registers.

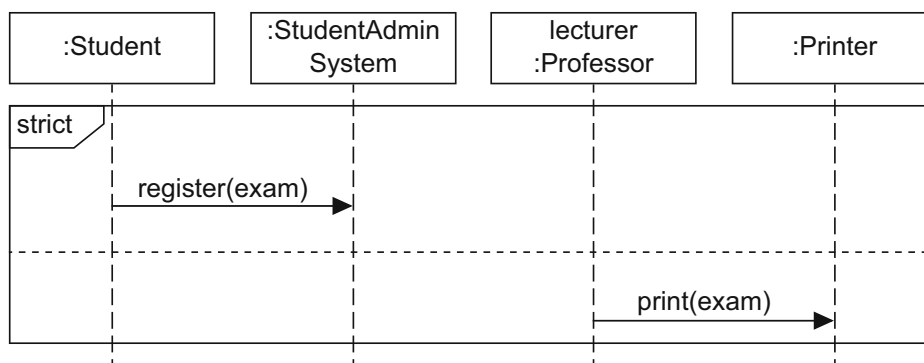
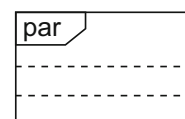


Figure 6.14
Example of a strict fragment

[Figure 6.15](#) contains the same message sequence as [Figure 6.13](#) but this time based on a strict order. This means that the messages are in a fixed order and there is only one trace.

The par fragment enables you to set aside any chronological order between messages in different operands. From a time perspective, the execution paths of the different operands can be interleaved as long as the restrictions of each individual operand are respected. Hence, the order of the different operands is irrelevant. However, this construct does not induce true parallelism, that is, it does not require simultaneous processing of the operands. This is contrary to what we would expect from the keyword “par” in par operators. Indeed, the par operator actually expresses *concurrency*, that is, the order of the events that are located in different operands is irrelevant. The par operator therefore has at least two operands. However, the order within an operand must be respected, meaning that there are local time axes for each operand and these must

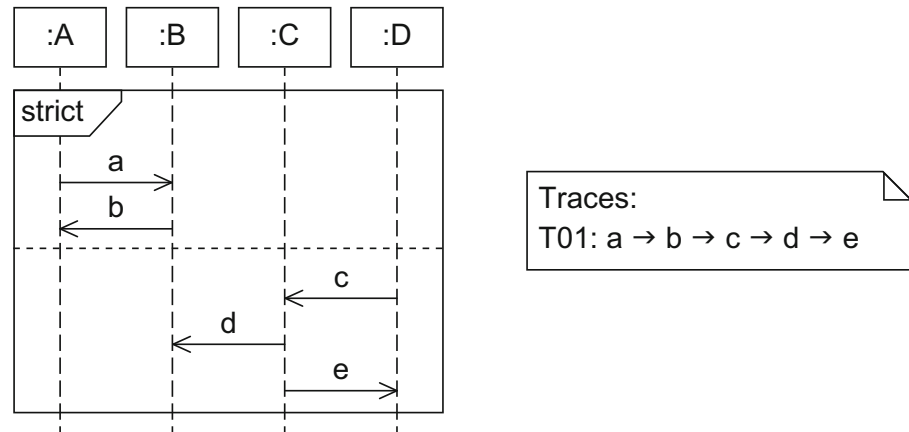
par fragment



Concurrent interaction

Figure 6.15

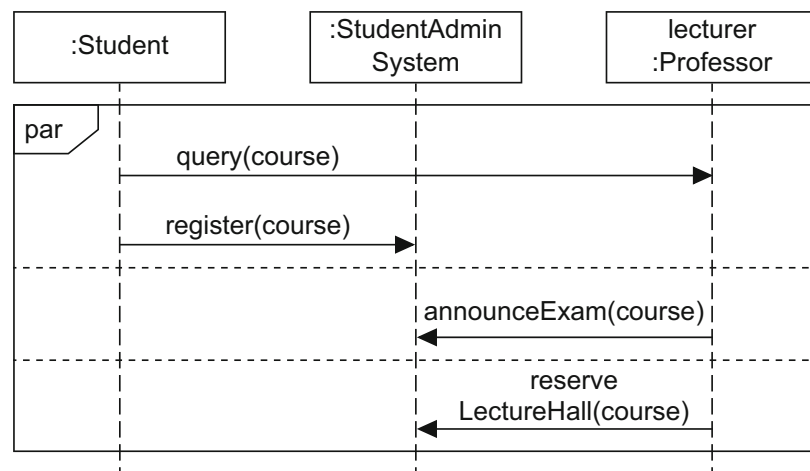
Traces in a strict fragment



be adhered to. [Figure 6.16](#) illustrates the use of par. At the beginning of a course, the lecturer has to complete certain activities. The lecturer must answer queries from students, announce exam dates, and reserve lecture halls. To do all of this, the lecturer has to communicate with different persons and systems. A par fragment is used to express that the order in which these activities are completed is irrelevant. What is important is that the default order between messages within an operand is adhered to, meaning that according to this sequence diagram, a student will never register for a course first and then send a query to the lecturer.

Figure 6.16

Example of a par fragment



[Figure 6.17](#) shows the possible traces for a concurrent interaction, whereby again the same message sequence as shown in [Figure 6.13](#) and [Figure 6.15](#) is used but this time with a par fragment. There is no longer any chronological connection between the messages from the different operands, which explains the multitude of possible traces. What is important is that the order of the messages within an operand is respected. For example, message a must always come before message b.

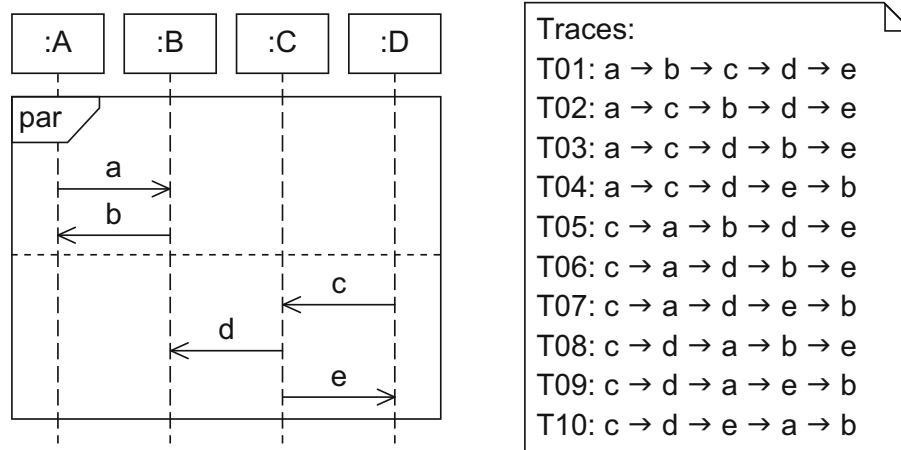


Figure 6.17
Traces in a par fragment

Alternatively, you can set aside the chronological order of events on a single lifeline using a *coregion*. This enables you to model concurrent events for a single lifeline. The order of event occurrences within a coregion is in no way restricted, even though they are arranged along the lifeline. The area of the lifeline to be covered by the coregion is marked by square brackets rotated by 90 degrees.

Coregion

A coregion can of course contain further combined fragments executed as a whole in any order. At the corresponding points, the revocation of the chronological order spreads out to the corresponding interaction partners of the lifeline with the coregion. The example modeled in Figure 6.16 with a par fragment is modeled in Figure 6.18 with a coregion. Semantically there is no difference between these two diagrams, meaning that no chronological order is defined for the messages that the lecturer sends and receives.

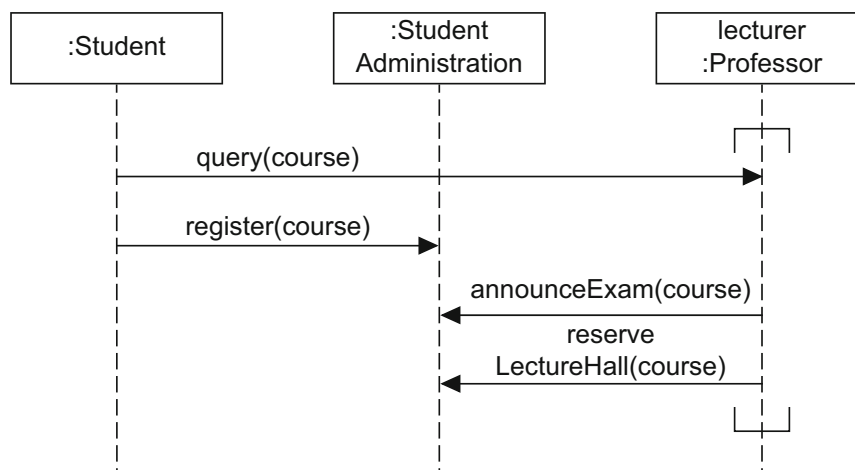
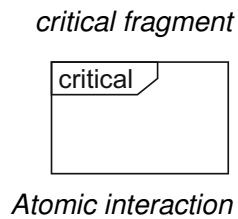
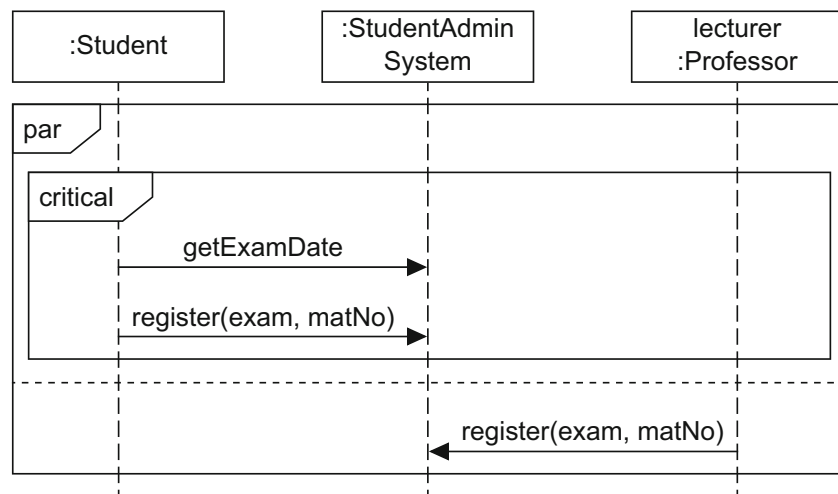


Figure 6.18
Example of the use of a coregion



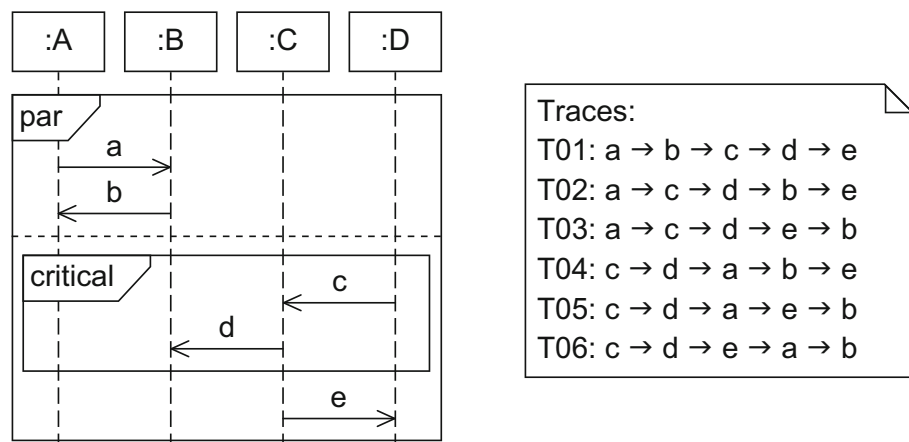
To make sure that certain parts of an interaction are not interrupted by unexpected events, you can use the critical fragment. This marks an atomic area in the interaction. Note that the standard order seq applies within a critical fragment. We can see an example of this in [Figure 6.19](#). The messages `getExamDate` and `register` are located in a critical fragment. This ensures that in the time between the request for an exam date and the actual registration, no message can occur that reserves the place shown as free for someone else. If the critical fragment was not present, the lecturer could execute another registration in the time between the request for free places and the registration by a student, thus taking the place away from the student.

Figure 6.19
Example of a critical
fragment



[Figure 6.20](#) is different to [Figure 6.17](#) only in the fact that messages `c` and `d` are enclosed by a critical fragment. This means that only those traces are valid in which message `d` immediately follows message `c`.

Figure 6.20
Traces in a critical
fragment



6.4.3 Filters and Assertions

Typically, a sequence diagram does not describe all aspects of an interaction. Some sequences are highlighted and explicitly declared as permitted traces. In most cases, however, there are further, permitted but not described traces that may occur. In some cases, you have to document all possible traces that may occur or document those that must not occur. In short, a sequence diagram contains valid, invalid, and unspecified traces. The combined fragments from the group “filters and assertions” define (i) which messages may occur but are not relevant for the description of the system, (ii) which messages must occur, and (iii) which messages must not occur. Unfortunately, the description of the fragments in this group is very compact in the UML standard, which is why in many situations, numerous questions about their exact meaning remain unanswered. Below we give a short breakdown of these fragments, basing our explanation as closely as possible on the standard.

Irrelevant messages are indicated by the ignore fragment, which expresses that these messages can occur at runtime but have no further significance for the functions depicted in the model. The irrelevant messages are noted in curly brackets after the keyword ignore. In [Figure 6.21\(a\)](#), the message status is contained in the set of irrelevant messages. It is used only to implement the server-client communication and is irrelevant for the presentation of the actual functionality.

ignore fragment
Irrelevant interaction

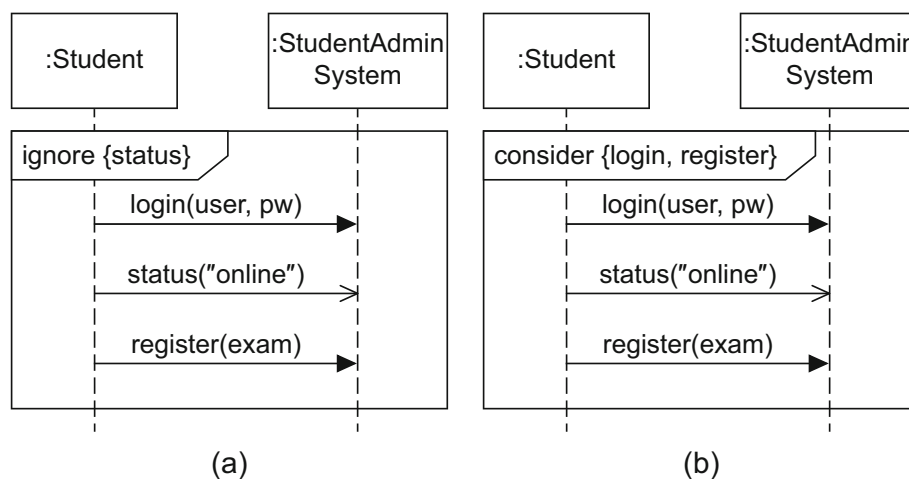


Figure 6.21
Examples of an ignore
fragment and a consider
fragment

In contrast, the consider fragment specifies those messages that are of particular importance for the interaction under consideration. These messages are also shown in set notation after the keyword. All messages that occur in the consider fragment but that are not specified in the set of relevant messages are automatically classified as irrelevant. They must

consider fragment
Relevant interaction

be treated as if they were listed as arguments of an ignore fragment (see Fig. 6.21(b), which is equivalent to Fig. 6.21(a)).

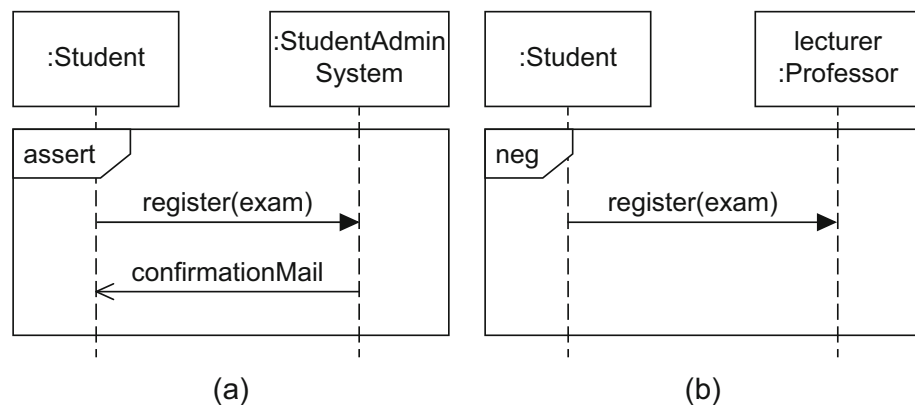
assert fragment
Asserted interaction

The assert fragment identifies certain modeled traces as mandatory. Deviations that occur in reality but that are not included in the diagram are not permitted. In effect, this means that the implementation requires precise modeling and the model is a complete specification. Figure 6.22(a) contains a corresponding example. When a student registers for an exam in the student administration system, the student receives an e-mail after the registration. If this sequence is not implemented precisely as specified, an error occurs.

neg fragment
Invalid interaction

With the neg fragment you model an invalid interaction, that is, you describe situations that must not occur. The neg fragment consists of exactly one operand. You can use this fragment to explicitly highlight frequently occurring errors and to depict critical, incorrect sequences. However, there is no limit to the number of possible interaction sequences that should/must not occur, and so you must not assume that using the neg fragment will cover all undesirable situations. Figure 6.22(b) expresses that a student may never register for an exam directly with the lecturer.

Figure 6.22
Examples of an assert
fragment and a neg
fragment



6.5 Further Language Elements

To enable us to specify interactions more precisely and to depict them more clearly, the sequence diagram offers the following additional language elements described in detail below. *Interaction references* and *continuation markers* enable you to break down sequence diagrams into modules to structure them more clearly. *Gates* allow you to model the flow of messages that takes place between different sequence diagrams

or combined fragments. *Parameters* and *local attributes* specify those values required for the execution of an interaction. *Time constraints* define when certain events must occur and *state invariants* specify conditions that are necessary for the execution of the interaction.

6.5.1 Interaction References

An *interaction reference* allows you to integrate one sequence diagram in another sequence diagram. On the one hand, this allows you to reuse interactions that you have already modeled, and on the other hand, it enables you to break down complex interaction sequences into modules and to depict them in simple form. Just like a combined fragment, an interaction reference is depicted in a rectangle with a pentagon in the upper left corner. The pentagon contains the keyword *ref*. The rectan-

Interaction reference

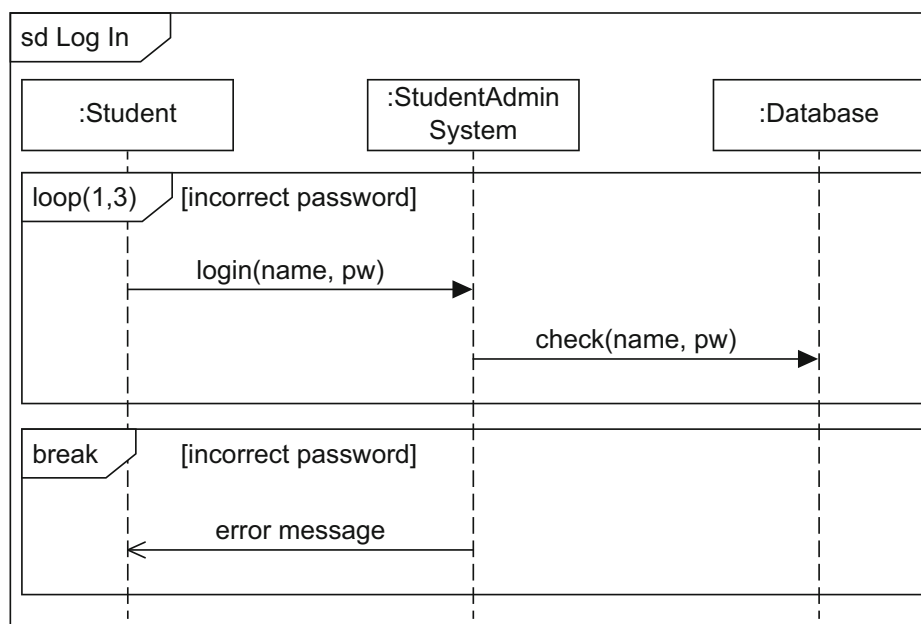
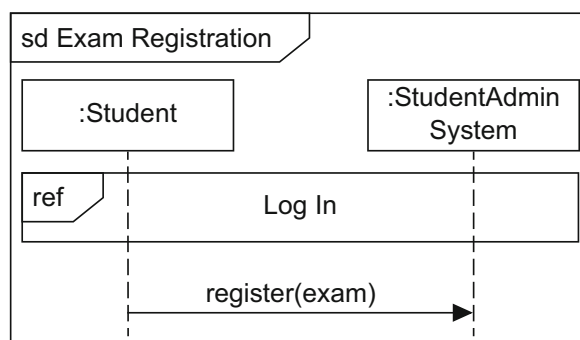


Figure 6.23
Example of an interaction reference

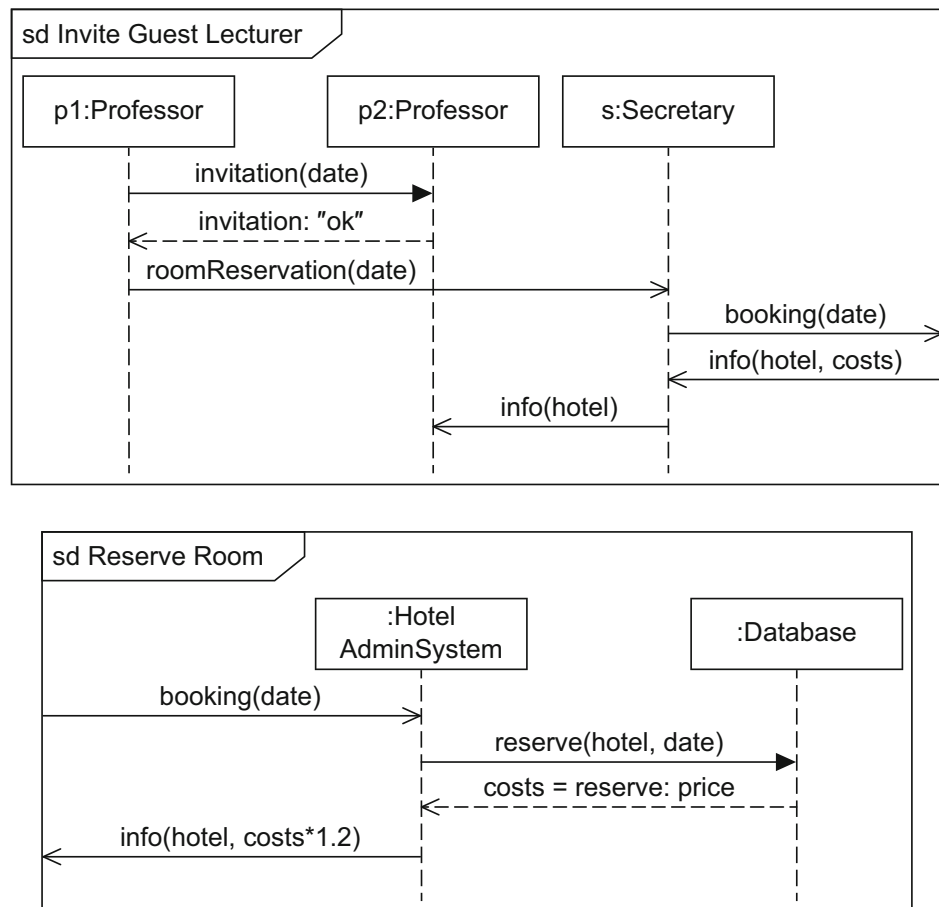


gle itself contains the name of the sequence diagram to be referenced, followed by optional arguments enclosed within parentheses, and an optional return value preceded by a colon. [Figure 6.23](#) illustrates the use of an interaction reference. The sequence for logging in to the student administration system is modeled in the sequence diagram Log In, which is referenced in the sequence diagram Exam Registration.

6.5.2 Gates

In principle, messages must not extend beyond the boundaries of an interaction, a referenced interaction, or a combined fragment, meaning that they must not exceed the frame arbitrarily. To enable you to extend the exchange of messages beyond such boundaries, UML offers *gates*. Using such gates thus allows you to send and receive messages beyond the boundaries of the interaction fragment. A gate is visualized by the tip or the end of a message arrow—depending on whether the

Figure 6.24
Example of a gate



message is an incoming or outgoing message—touching the boundary of the frame that represents the sequence diagram, the interaction reference, or the combined fragment. Gates are identified either by a name or by the name of a message that uses the gate, optionally together with the direction of the message (e.g., booking in [Fig. 6.24](#)). They allow you to define a specific sender and specific receiver for each message, even if the sender or receiver is outside the respective interaction or outside the fragment. You do not have to include gates explicitly for combined fragments, meaning that a message may point directly to the receiver.

6.5.3 Continuation Markers

Continuation markers allow you to modularize the operands of an alt fragment. This enables you to break down complex interactions into parts and connect them to one another with markers. Here, a *start marker* at the end of an interaction part points to a *target marker* with the same name at the beginning of another external interaction part. Continuation markers are denoted within rectangles with rounded corners that can extend across multiple interaction partners. Continuation markers with the same name must refer to the same interaction partners. [Figure 6.25](#) and [Figure 6.26](#) model the example from [Figure 6.10](#) on page 117 with three different start markers ([Fig. 6.25](#)) and the respective corresponding target markers ([Fig. 6.26](#)). If, for example, the start marker OK in [Figure 6.25](#) is reached, the sequence diagram continues with the interactions detailed under the target marker with the same name in [Figure 6.26](#). There is no return to the start marker—in contrast to an interaction reference, which can be compared to a macro. A continuation marker can also be the only element of an operand, thus increasing the clarity of the diagram. You can assign multiple start markers to one target marker. The target and start markers do not have to be located in the same sequence diagram.

Continuation marker

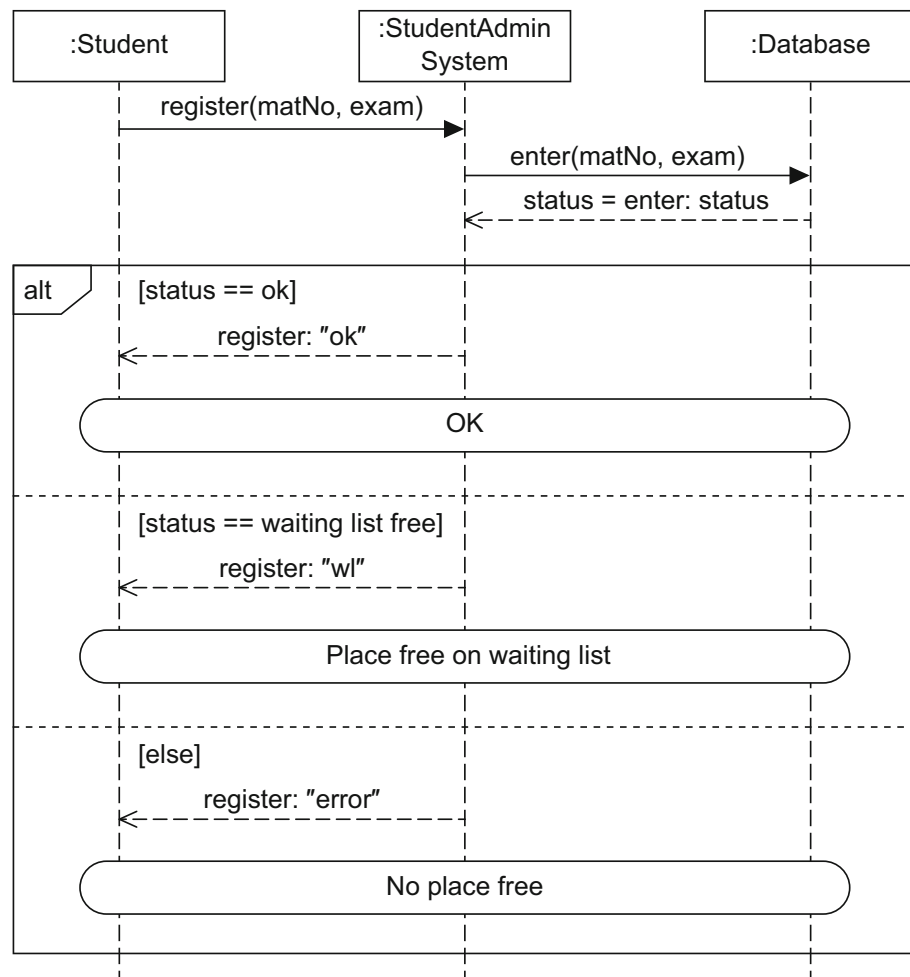
Start marker and target marker

6.5.4 Parameters and Local Attributes

Just like all of the other types of diagrams in UML 2.4.1, the sequence diagram is enclosed by a rectangular frame with a small pentagon in the upper left corner. This pentagon contains the keyword `sd` to clearly indicate that the content of the rectangle is a sequence diagram. The keyword `sd` is followed by the name of the sequence diagram and optional *parameters* separated by commas and enclosed within parentheses. You

Figure 6.25

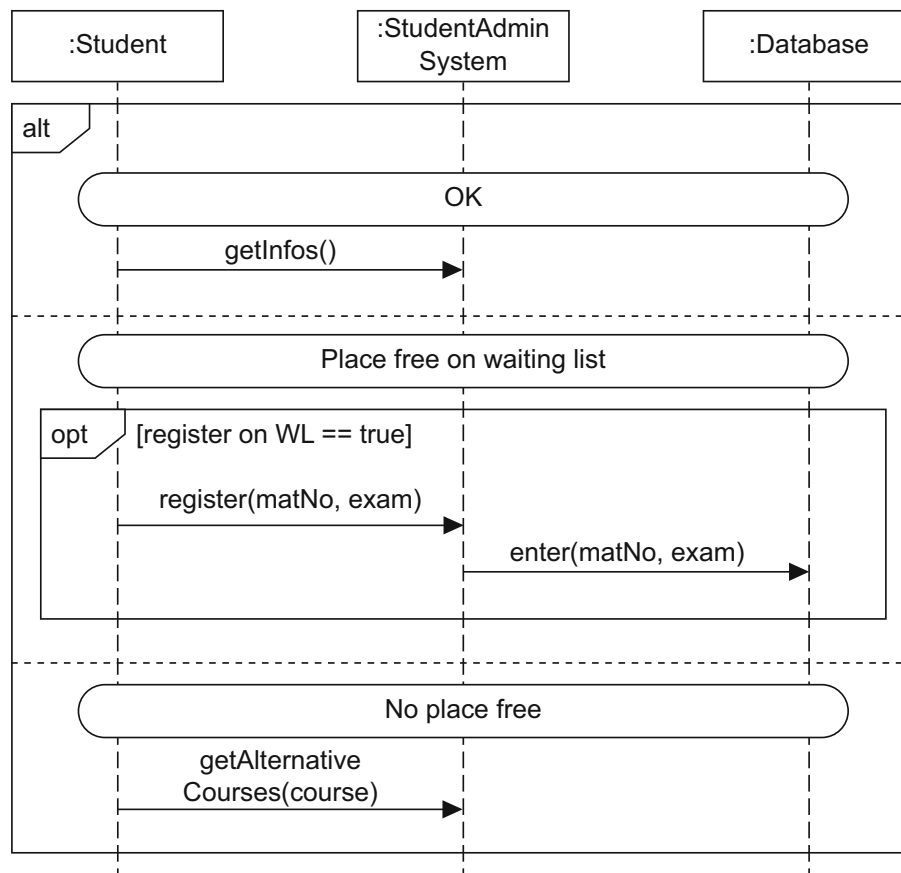
Example of a continuation marker (start markers)



can declare *local attributes* at any point in the diagram, whereby the syntax of the attributes corresponds to the attribute specifications in the class diagram (see Fig. 4.5 on page 54). Alternatively, you can declare local attributes in a note.

6.5.5 Time Constraints

<i>Time constraint</i>	<i>Time constraints</i> specify either the time at which events occur or a time period between two events. Time constraints are noted in curly brackets.
<i>Timing expression</i>	The <i>timing expression</i> represents either a concrete time specification, for example, {12:00}, or a calculation rule, such as {12:00+d}. You can specify <i>absolute times</i> with the keyword <i>at</i> , for example, {at(12:00)}. <i>Relative times</i> are specified with reference to a starting event using the keyword <i>after</i> , for example, {after(5sec)}. In both cases, the timing expression is denoted within curly brackets.

**Figure 6.26**

Example of a continuation marker (target markers)

You can also specify time intervals. Again, an interval is enclosed by curly brackets and contains an expression in the form lower limit..upper limit. To express that an event takes place between 12:00 and 13:00, for example, you would use the form { 12:00..13:00 }.

The keyword *now* specifies the current time. It can be assigned to any attribute, for example, *t=now*. Naturally, you can use this attribute in any time constraints, for example, { *t.t+5* }.

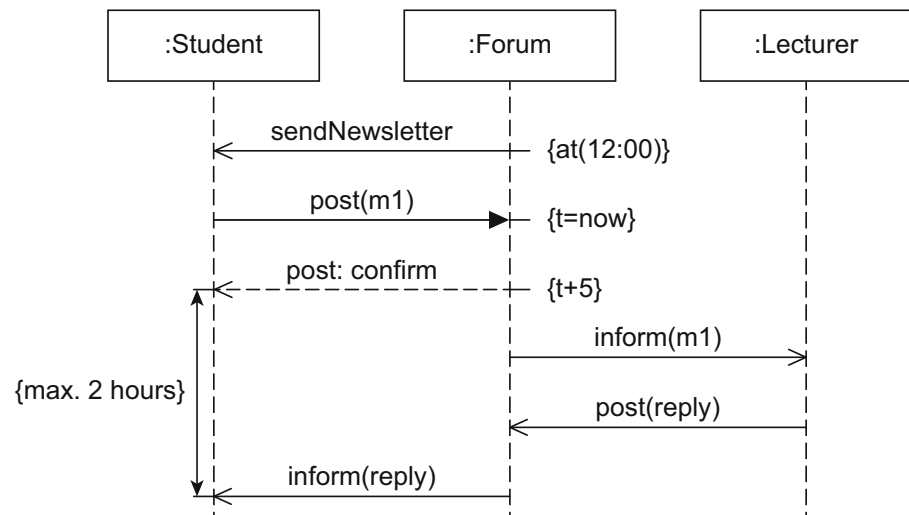
The calculation of the duration of a message transmission is indicated by the keyword *duration*.

You assign a time constraint to an event using a *timing mark* represented by a short horizontal line in the diagram. If a time constraint refers to two events, meaning that the duration between two events is to be defined, you specify this using two timing marks. Figure 6.27 shows some examples of time constraints. The diagram involves communication between students and lecturers via a forum. The forum sends a newsletter to the students at 12:00. At time *t*, a student posts a message *m1*. Five time units later, the student receives notification that the message is being posted. A maximum of two hours later, the response from the lecturer arrives.

Interval

Timing mark

Figure 6.27
Examples of time constraints

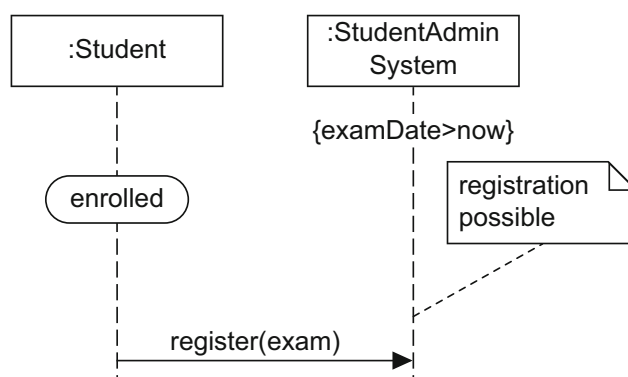


6.5.6 State Invariants

State invariant You can specify *state invariants* for an interaction. A state invariant asserts a certain condition must be fulfilled at a certain time. It is always assigned to a specific lifeline. The evaluation of whether the invariant is true takes place before the subsequent event occurs. If the state invariant is not true, either the model or the implementation is incorrect. State invariants can reference states from the related state machine diagram or they can be logical conditions that refer to local attributes.

UML offers three notation alternatives for state invariants: you can specify a state invariant within curly brackets directly on a lifeline; you can attach it as a note; or you can also place it in a rectangle with rounded edges at the corresponding point of the lifeline. The three notation alternatives are shown in Figure 6.28. A student can only register for an exam (i) if the student is enrolled, (ii) if the exam has not yet taken place, and (iii) if registration for the exam is possible.

Figure 6.28
Notation alternatives for state invariants



6.6 Creating a Sequence Diagram

Instead of a final example, in this section we will look at two scenarios that you can use a sequence diagram for. In particular, we will illustrate the connection between the class diagram and the sequence diagram. Then we will conclude the section with a typical application for sequence diagrams, namely the description of design patterns [20].

6.6.1 The Connection between a Class Diagram and a Sequence Diagram

We have repeatedly stated that the different UML diagrams should not be considered independently of one another; they merely offer different views of a certain content. For example, the class diagram shown in [Figure 6.29](#) models a part of a university system that also includes the student administration system.

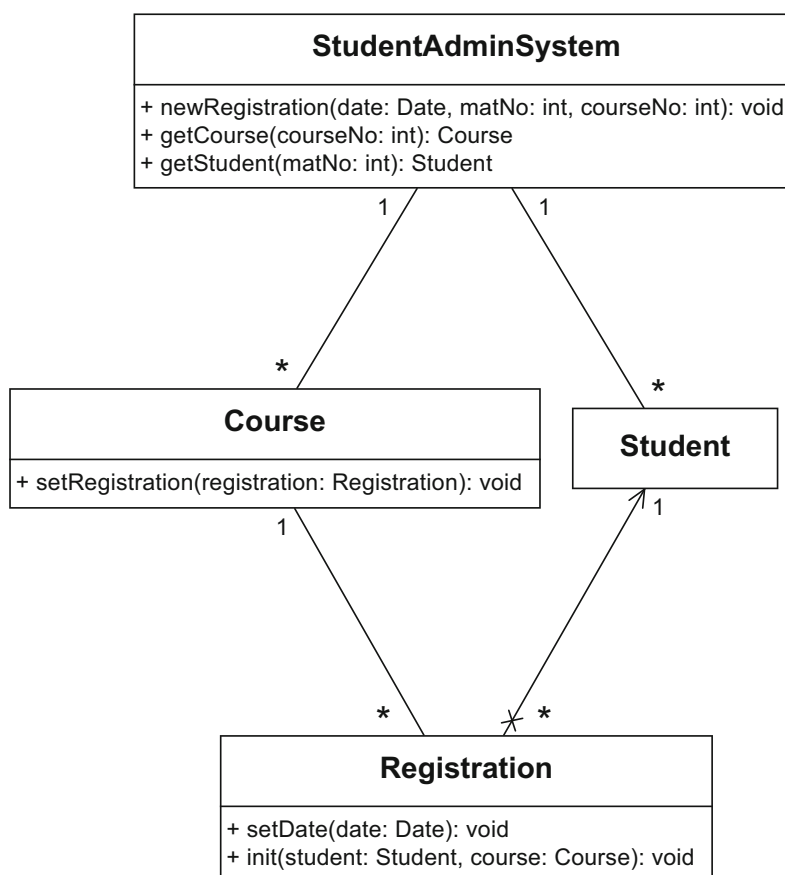
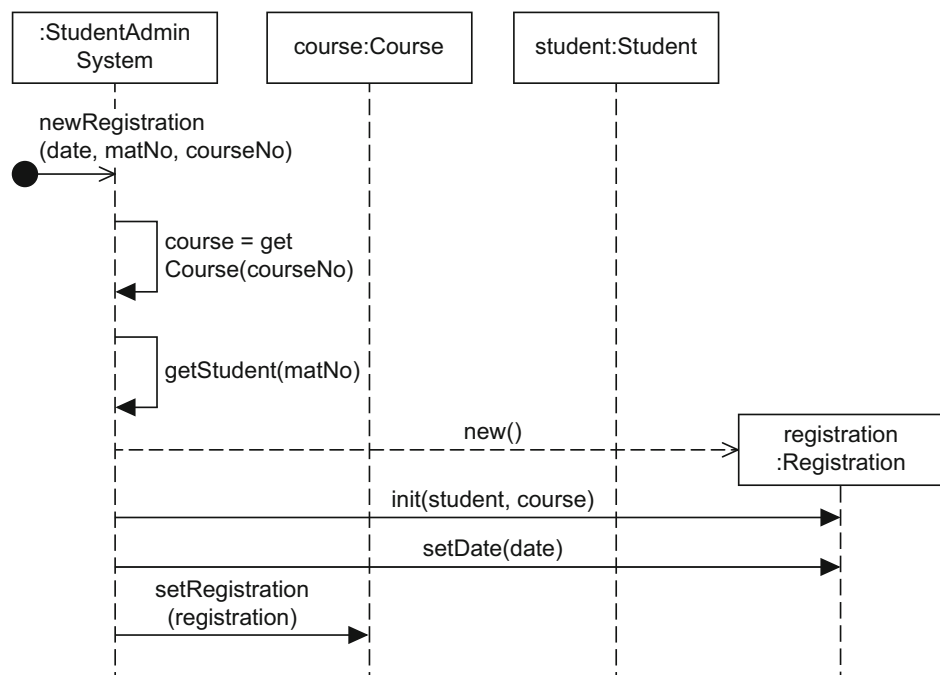


Figure 6.29
Class diagram

The student administration system has direct access to all students and courses. The system knows the registration data for students and courses that is stored in the class `Registration`.

We want to depict the communication that is required to create a new registration of a certain student for a certain course. To do this, the method `newRegistration` of the class `StudentAdminSystem` must be called. To create a new registration, we have to know the student object that belongs to the respective matriculation number and the course object that belongs to the given course number. We can obtain these by calling the operations `getCourse` and `getStudent`. As soon as we have this information, we can create a new object of the type `Registration` and call the `init` operation that sets the student and the course for the registration object. Now we just have to establish the connection between the registration and the course, as navigability in both directions is assumed. We do this by calling the method `setRegistration`. We do not have to do this for the registration and student objects, as navigation from `Student` to `Registration` is not possible. The resulting sequence diagram is shown in [Figure 6.30](#).

Figure 6.30
Sequence diagram based
on class diagram



6.6.2 Describing Design Patterns

Sequence diagrams are often used to describe design patterns. Design patterns offer solutions for describing recurring problems. In the following we will look at the Transfer Object Assembler pattern [18], which describes what happens when a client in a distributed environment requires information from three business objects.

In the solution shown in [Figure 6.31](#), the client requires knowledge about the three business objects in order to access the required data. The client is therefore strongly linked to the business objects—which is generally not desirable. We can use the Transfer Object Assembler pattern to break down these dependencies. In this pattern, an assembler merges the data from multiple business objects into one transfer object that is then transferred to the client. The client thus receives the required data in an encapsulated form. In concrete terms, the pattern is implemented as described below (see [Fig. 6.32](#)).

Modeling the communication of the Transfer Object Assembler pattern

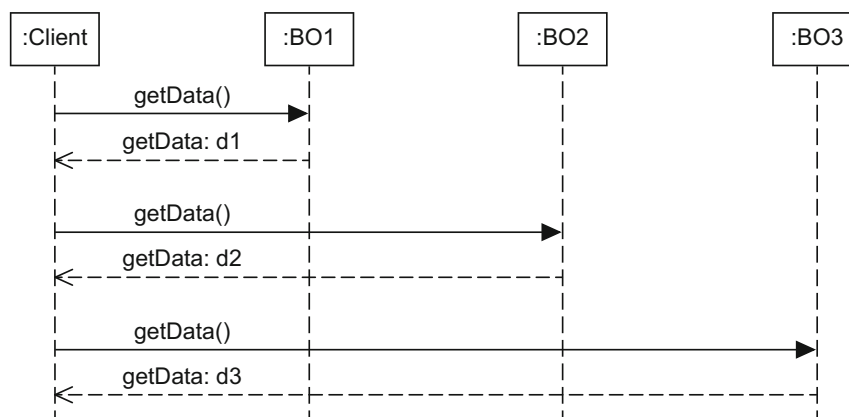
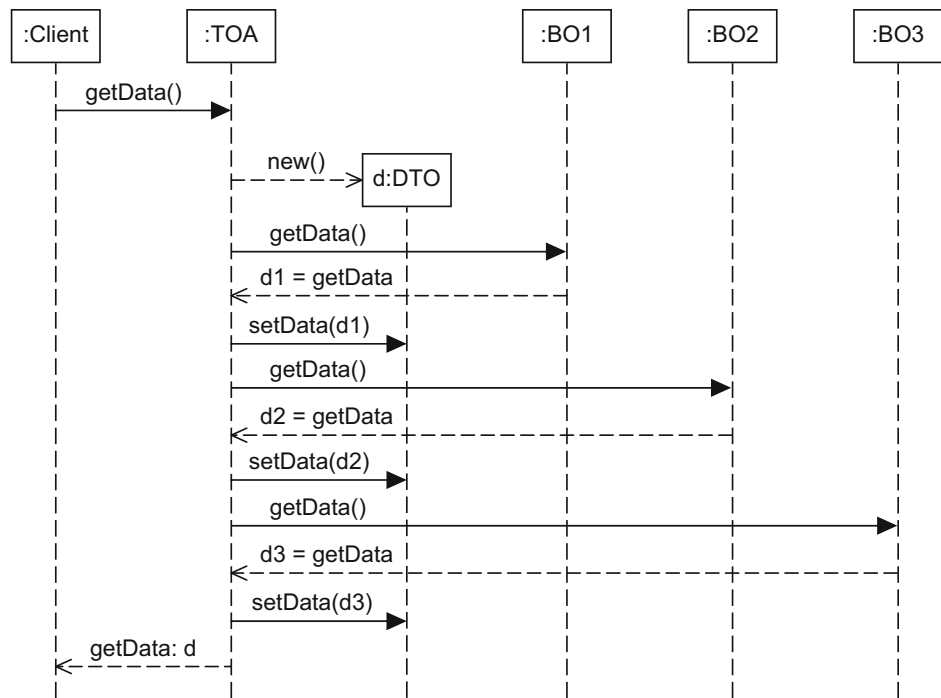


Figure 6.31
Application scenario for
Transfer Object Assembler
pattern

Using `getData()` the client requests the required information from an object of type `TransferObjectAssembler` (TOA). The TOA object creates the object `d` of type `DataTransferObject` (DTO) and fills it with the data from the three different business objects (BO1, BO2, BO3). The data of a business object can be queried using `getData()`. The object `d` offers `setData` methods that allow the entry of data. Finally, the object of type `TransferObjectAssembler` returns `d` to the client.

Figure 6.32

Description of the Transfer Object Assembler pattern using a sequence diagram



6.7 The Communication, Timing, and Interaction Overview Diagrams

In addition to the sequence diagram, UML supports three further types of interaction diagrams:

- Communication diagram
- Timing diagram
- Interaction overview diagram

The four types of interaction diagrams of UML are generally equivalent for simple interactions as they are based on the same basic elements. With the specification of the communication partners involved and the messages exchanged, they all describe certain communication situations. However, the focus is different for each type of diagram.

In this section, we briefly compare the four interaction diagrams in examples. The examples are illustrated in [Figures 6.33 to 6.36](#), showing various aspects of the communication between a student and the e-learning system of a university.

[Figure 6.33](#) shows the log-in process as a sequence diagram. There are three interaction partners: student, e-learning system, and database. The student wants to log in to the system and therefore sends a corresponding message to the e-learning system. A query to the database verifies the access rights and the student receives a positive response

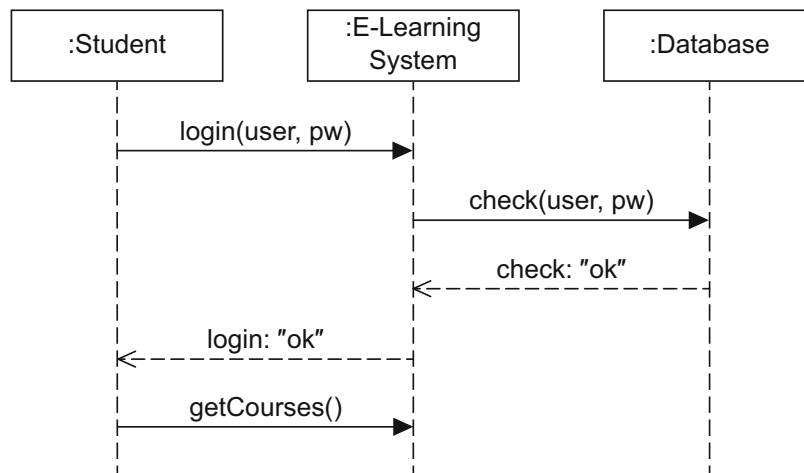


Figure 6.33
Example of sequence diagram

message. Any possible error cases are not considered. The student then requests the list of subscribed courses.

The *communication diagram* is a structured diagram that models the relationships between communication partners. It therefore shows directly who communicates with whom. The relationships are the result of the exchange of messages. Here, time is not a separate dimension. The order in which the messages are exchanged is expressed using decimal classification (sequential numbering) for the messages. Figure 6.34 shows the log-in process as a communication diagram. Again, the interaction partners are student, e-learning system, and database. The diagram shows that the student communicates with the e-learning system twice: once using `login(user, pw)` and once using `getCourses()`. It also shows that the e-learning system communicates with the database using `check(user, pw)`. The numbering results in the order login, check, and get-Courses. All three messages are synchronous messages, as shown by the arrowheads; asynchronous messages would be shown with open arrowheads as in a sequence diagram. Response messages are not depicted in the communication diagram.

Communication diagram

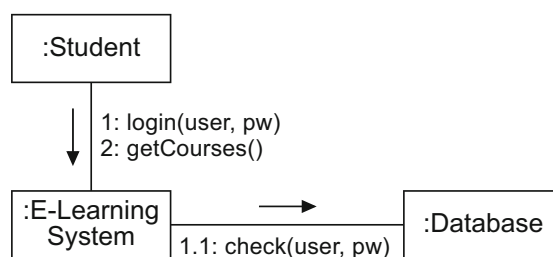
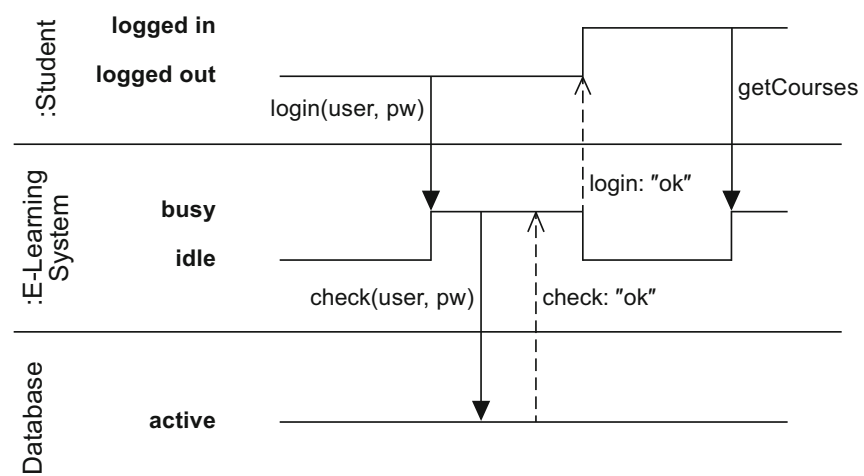


Figure 6.34
Example of communication diagram

Timing diagram

The *timing diagram* shows state changes of the interaction partners that result from the occurrence of events. In contrast to the sequence diagram, in which the arrangement of the interaction partners and the time axis is exactly the opposite, in the timing diagram the interaction partners are listed on the vertical axis and the horizontal axis represents the chronological order. In the timing diagram, lifelines are depicted by a whole area in which states and state transitions can be represented. The name of the lifeline (role name and/or class) is noted vertically at the left boundary of the area. [Figure 6.35](#) thus shows the interaction partners student, e-learning system, and database. A student can be in the state logged in or logged out and the e-learning system can take the states idle or busy. For the database there is only the state active. If the student now sends the message login(user, pw) to the e-learning system, the system changes from the state idle to the state busy and sends the message check(user,pw) to the database. The database verifies the data and thus the student is allowed access to the system. The student is informed of this with a corresponding response message. The student is now in the state logged in. The e-learning system can briefly return to the state idle until the student sends the getCourses request.

Figure 6.35
Example of timing diagram



*Interaction overview
diagram*

The *interaction overview diagram* shows the different interactions and visualizes the order in and conditions under which they take place. This allows you to place various interaction diagrams in a logical order. To do this, you use primarily the concepts of the activity diagram (see Chapter 7). Instead of specifying nodes for actions and objects, you specify entire interaction diagrams and interaction references as nodes which you can then place in order using the control structures of the activity diagram. A solid circle represents the initial node and a solid circle with a surrounding circle represents a final node. You can implement different paths using decision nodes represented by a hollow

diamond. [Figure 6.36](#) shows an interaction overview diagram that, in addition to an initial and final node, also contains this type of branch node. If the student has the necessary authorization, the student can execute the interaction Forum, represented as a sequence diagram.

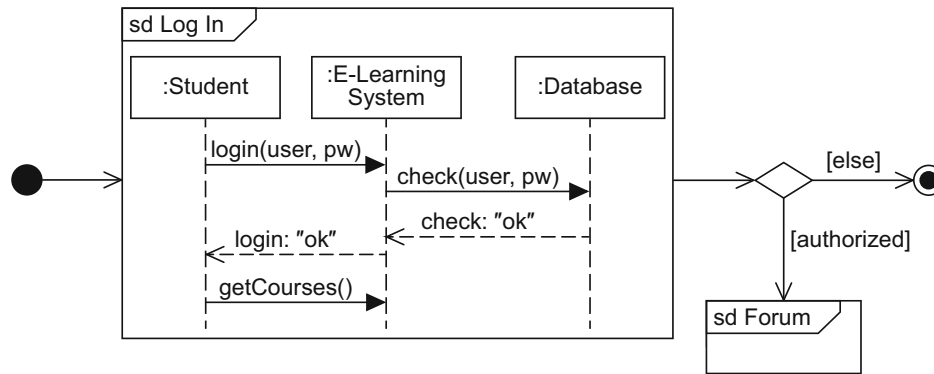


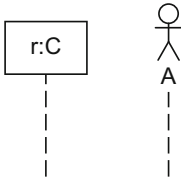
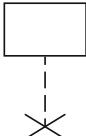
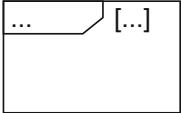
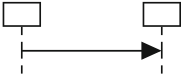
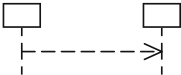

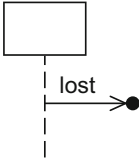
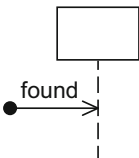
Figure 6.36
Example of interaction
overview diagram

6.8 Summary

The sequence diagram is one of four interaction diagrams in UML. Interaction diagrams model the communication between different interaction partners, whereby each of the four diagrams focuses on a different aspect. In practice, the sequence diagram is the most frequently used of the interaction diagrams. The presentation of communication protocols and design patterns are particularly prominent applications of sequence diagrams as they enable a compact and clear specification. In addition to the interaction partners, which are depicted in the form of lifelines, the sequence diagram contains different types of messages (synchronous, asynchronous, response message, create message). The chronological order of the messages is generally assumed to be from top to bottom along the vertical line. Twelve types of combined fragments provide you with different control structures that enable you to control the interaction. The most important elements of the sequence diagram are summarized in [Table 6.2](#).

Table 6.2

Notation elements for the sequence diagram

<i>Name</i>	<i>Notation</i>	<i>Description</i>
Lifeline		Interaction partners involved in the communication
Destruction event		Time at which an interaction partner ceases to exist
Combined fragment		Control constructs
Synchronous message		Sender waits for a response message
Response message		Response to a synchronous message
Asynchronous message		Sender continues its own work after sending the asynchronous message
Lost message		Message to an unknown receiver
Found message		Message from an unknown sender